# BITS Pilani presentation

BITS Pilani
Pilani Campus

A. A. Nippun Kumaar

# AIML ZG512 - Deep Reinforcement learning
# Markov Decision Processes

# Agenda for the class

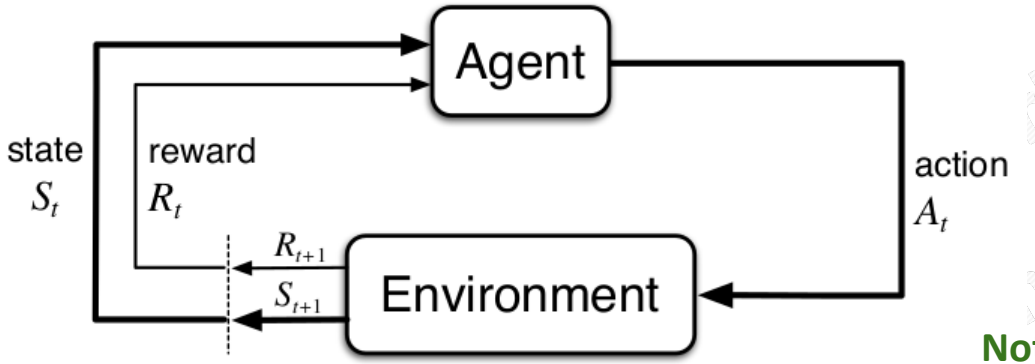


- Agent-Environment Interface (Sequential Decision Problem)
- MDP
  - Defining MDP,
  - Rewards,
  - Returns, Policy & Value Function,
  - Optimal Policy and Value Functions
- Approaches to solve MDP

Acknowledgements: Some of the slides were adopted *with permission* from the course CSCE-689 (Texas A&M University) by Prof. Guni Sharon

# Agent-Environment Interface

- Agent - Learner & the decision maker
- Environment - Everything outside the agent
- Interaction:
  - Agent performs an action
  - Environment responds by
    - presenting a new situation (change in state)
    - presents numerical reward
- Objective (of the interaction):
  - Maximize the return (cumulative rewards) over time




$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$

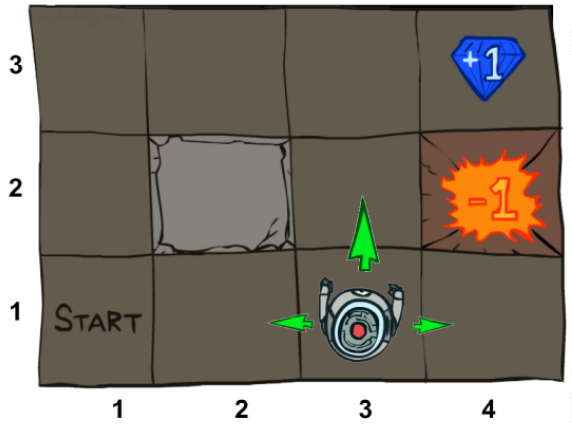
**Note**:

- Interaction occurs in discrete time steps

# Grid World Example



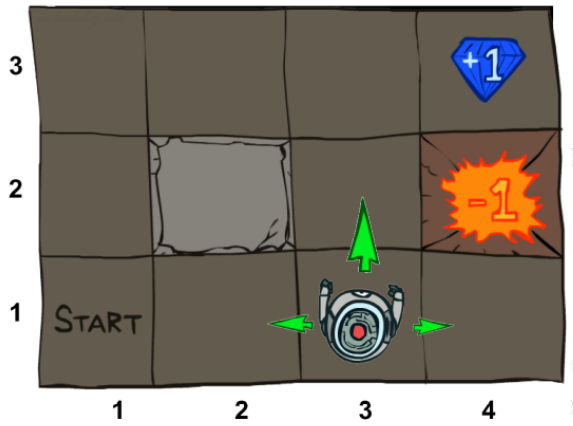
- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North (if there is no wall there)
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - -0.1 per step (battery loss)
  - +1 if arriving at (4,3) ; -1 for arriving at (4,2) ;1 for arriving at (2,2)
- Goal: maximize accumulated rewards



# Markov Decision Processes



- An MDP is defined by
  - A set of states
  - A set of actions
  - State-transition probabilities
    - Probability of arriving to after performing at
    - Also called the model dynamics
  - A reward function
    - The utility gained from arriving to after performing at
    - Sometimes just or even
  - A start state
  - Maybe a terminal state



# Markov Decision Processes



## Model Dynamics

$$p(s', r \mid s, a) \doteq \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}$$

## State-transition probabilities

$$p(s' \mid s, a) \doteq \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r \mid s, a)$$

## Expected rewards for state–action–next-state triples

$$r(s, a, s') \doteq \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in \mathcal{R}} r \frac{p(s', r \mid s, a)}{p(s' \mid s, a)}$$

# Markov Decision Processes - Discussion



- *MDP framework is abstract and flexible*
  - Time steps need not refer to fixed intervals of real time
  - The actions can be
    - at low-level controls or high-level decisions
    - totally mental or computational
  - States can take a wide variety of forms
    - Determined by *low-level sensations* or *high-level and abstract* (ex. symbolic descriptions of objects in a room)
- *The agent–environment boundary represents the limit of the agent's absolute control*, not of its knowledge.
  - *The boundary can be located at different places for different purposes*

# Markov Decision Processes - Discussion



- *MDP framework is a considerable abstraction of the problem of goal-directed learning from interaction.*
- It proposes that *whatever the details of the sensory, memory, and control apparatus, and whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment:*
  - *one signal to represent the choices made by the agent (the actions)*
  - *one signal to represent the basis on which the choices are made (the states),*
  - *and one signal to define the agent's goal (the rewards).*

# MDP Formalization : Video Games



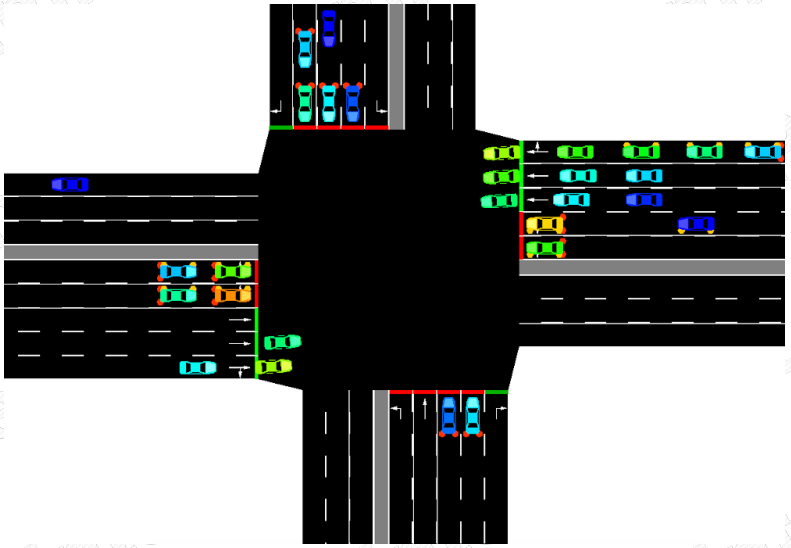
- *State:*
  - *raw pixels*
- *Actions:*
  - *game controls*
- *Reward:*
  - *change in score*
- *State-transition probabilities:*
  - *defined by stochasticity in game evolution*

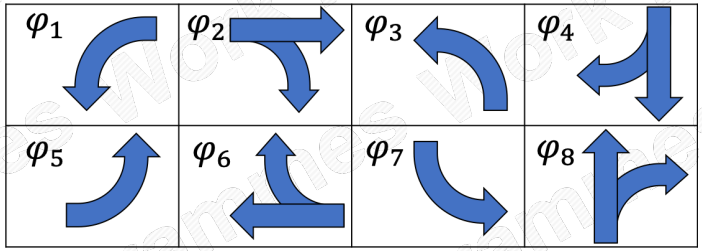
**Ref**: Playing Atari with deep reinforcement learning", Mnih et al., 2013

# MDP Formalization : Traffic Signal Control



- *State*:
  - *Current signal assignment (green, yellow, and red assignment for each phase)*
  - *For each lane: number of approaching vehicles, accumulated waiting time, number of stopped vehicles, and average speed of approaching vehicles*
- *Actions*:
  - *signal assignment*
- *Reward*:
  - *Reduction in traffic delay*
- *State-transition probabilities*:
  - *defined by stochasticity in approaching demand*






**Ref**: "Learning an Interpretable Traffic Signal Control Policy", Ault et al., 2020

# MDP Formalization : Recycling Robot (Detailed Ex.)



- *Robot has*
  - *sensors for detecting cans*
  - *arm and gripper that can pick the cans and place in an onboard bin;*
- *Runs on a rechargeable battery*
- *Its control system has components for interpreting sensory information, for navigating, and for controlling the arm and gripper*
- *Task for the RL Agent: Make high-level decisions about how to search for cans based on the current charge level of the battery*

# MDP Formalization : Recycling Robot (Detailed Ex.)



- *State:*
  - *Assume that only two charge levels can be distinguished*
  - *S = {high, low}*
- *Actions:*
  - *A(high) = {search, wait}*
  - *A(low) = {search, wait, recharge}*
- *Reward:*
  - *Zero most of the time, except when securing a can*
  - *Cans are secured by searching and waiting, but* $r_{search} > r_{wait}$
- *State-transition probabilities:*
  - *[Next Slide]*

# MDP Formalization : Recycling Robot (Detailed Ex.)



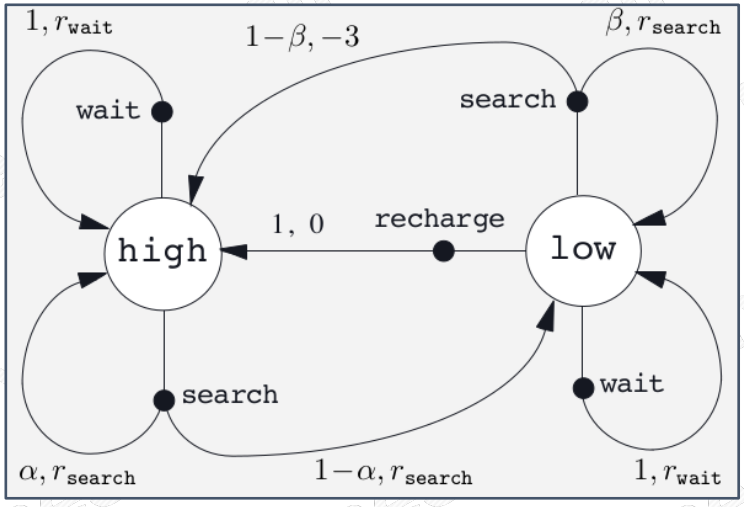
- *State-transition probabilities (contd…):*

| $s$ | $a$ | $s'$ | $p(s'\|s,a)$ | $r(s,a,s')$ |
|---|---|---|---|---|
| high | search | high | $\alpha$ | $r_{\texttt{search}}$ |
| high | search | low | $1-\alpha$ | $r_{\texttt{search}}$ |
| low | search | high | $1-\beta$ | $-3$ |
| low | search | low | $\beta$ | $r_{\texttt{search}}$ |
| high | wait | high | 1 | $r_{\texttt{wait}}$ |
| high | wait | low | 0 | - |
| low | wait | high | 0 | - |
| low | wait | low | 1 | $r_{\texttt{wait}}$ |
| low | recharge | high | 1 | 0 |
| low | recharge | low | 0 | - |

# MDP Formalization : Recycling Robot (Detailed Ex.)

- *State-transition probabilities (contd…):*

| $s$ | $a$ | $s'$ | $p(s' \mid s, a)$ | $r(s, a, s')$ |
|---|---|---|---|---|
| high | search | high | $\alpha$ | $r_{\text{search}}$ |
| high | search | low | $1-\alpha$ | $r_{\text{search}}$ |
| low | search | high | $1-\beta$ | $-3$ |
| low | search | low | $\beta$ | $r_{\text{search}}$ |
| high | wait | high | 1 | $r_{\text{wait}}$ |
| high | wait | low | 0 | - |
| low | wait | high | 0 | - |
| low | wait | low | 1 | $r_{\text{wait}}$ |
| low | recharge | high | 1 | 0 |
| low | recharge | low | 0 | - |



# Note on Goals & Rewards

- *Reward Hypothesis*:

  *All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*

- *The rewards we set up truly indicate what we want accomplished,*
  - *not the place to impart prior knowledge on how we want it to do*
- *Ex: Chess Playing Agent*
  - *If the agent is rewarded for taking opponents pieces, the agent might fall for the opponent's trap.*
- *Ex: Vacuum Cleaner Agent*
  - *If the agent is rewarded for each unit of dirt it sucks, it can repeatedly deposit and suck the dirt for larger reward*

# Returns & Episodes

- *Goal is to maximize the expected return*
- *Return (Gt) is defined as some specific function of the reward sequence*
- *Episodic tasks vs. Continuing tasks*
- *When there is a notion of final time step, say T, return can be*

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

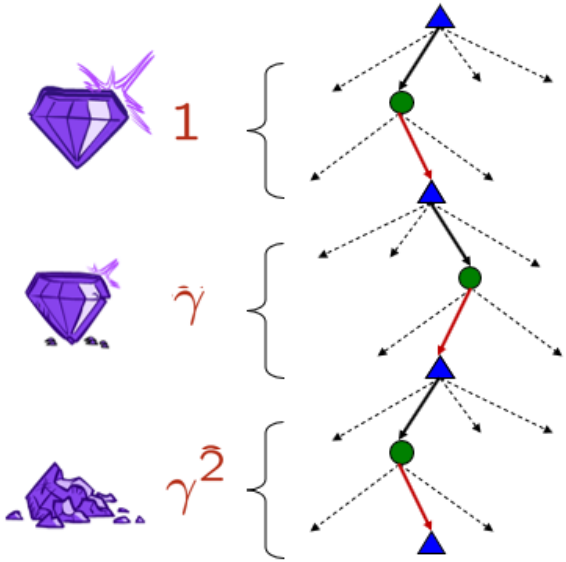
  - *Applicable when agent-environment interaction breaks into episodes*
  - *Ex: Playing Game, Trips through maze etc. [ called episodic tasks]*

# Returns & Episodes

- *Generally T = ∞*
  - *What if the agent receive a reward of +1 for each timestep?*
  - *Discounted Return:*

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

**Note:** $\gamma$ is a parameter, $0 \leq \gamma \leq 1$, called the *discount rate*.




  - *Discount rate determines the present value of future rewards*

# Returns & Episodes

- *What if γ is 0?*
- *What if γ is 1?*
- *Computing discounted rewards incrementally*

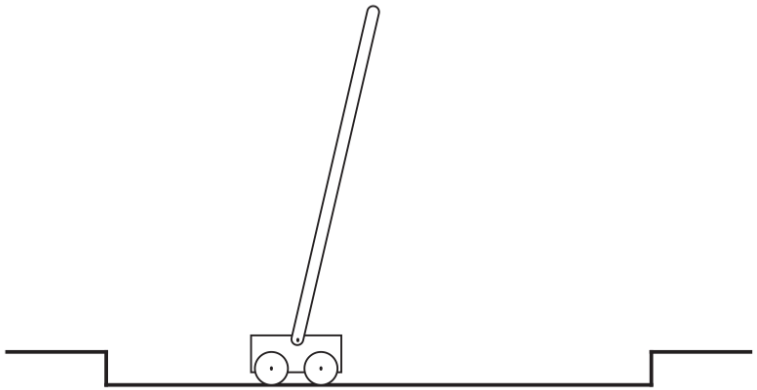
$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots\right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

- *Sum of an infinite number of terms, it is still finite if the reward is nonzero and constant and if γ < 1.*
- *Ex: reward is +1 constant*

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$$

# Returns & Episodes

- *Objective: To apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over*
- *Discuss:*
  - *Consider the task as episodic, that is try/maintain balance until failure. What could be the reward function?*
  - *Repeat prev. assuming task is continuous.*



# Policy

- A mapping from states to probabilities of selecting each possible action.
  - $\pi$ (a|s) is the probability that $A_t = a$ if $S_t = s$
- The purpose of learning is to improve the agent's policy with its experience

# Defining Value Functions

**State-value function for**

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \text{ for all } s \in \mathcal{S}$$

**Action-value function for**

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

# Defining Value Functions

**State Value function in terms of Action-value function for policy π**

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) q_\pi(s,a)$$

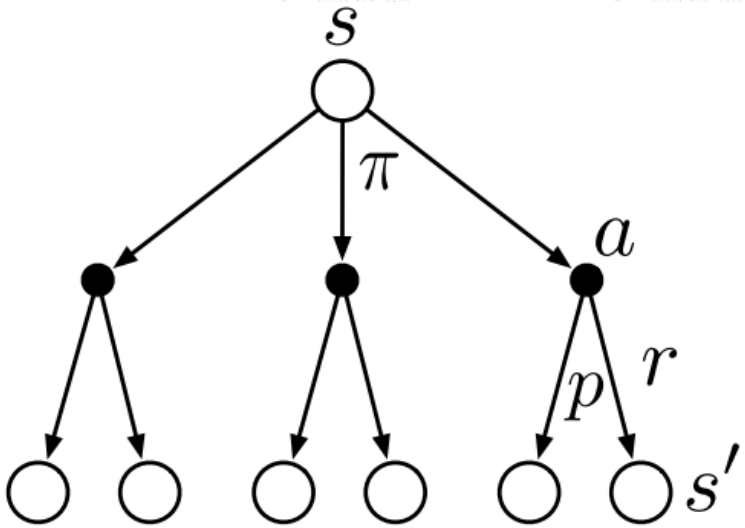
**Action Value function in terms of State value function for policy π**

$$q_\pi(s,a) = \sum_{s \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a) \left[ r + \gamma v_\pi(s') \right]$$

# Bellman Equation for $V_\pi$

$$v_\pi(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]$$

**Value of the start state must equal**
**(1) the (discounted) value of the expected next state,**
**plus**
**(1) the reward expected along the way**



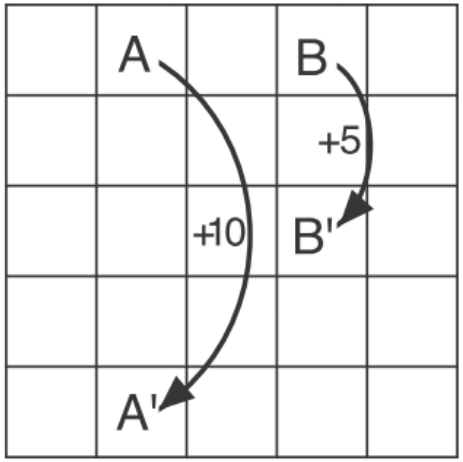

**Backup Diagram**

# Understanding $V_\pi(s)$ with Gridworld

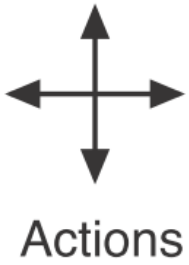


**Reward**:

- -1 if an action takes agent off the grid
- Exceptional reward from A and B for all actions taking agent to A' and B' resp.
- 0, everywhere else




**Exceptional reward dynamics**



Actions

| | | | | |
|---|---|---|---|---|
| 3.3 | 8.8 | 4.4 | 5.3 | 1.5 |
| 1.5 | 3.0 | 2.3 | 1.9 | 0.5 |
| 0.1 | 0.7 | 0.7 | 0.4 | -0.4 |
| -1.0 | -0.4 | -0.4 | -0.6 | -1.2 |
| -1.9 | -1.3 | -1.2 | -1.4 | -2.0 |

**State-value function for the equiprobable random policy with $\gamma = 0.9$**

# Understanding $V_\pi(s)$ with Gridworld



$$v_\pi(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]$$

**Verify** $V_\pi(s)$ using Bellman equation for this state with $\gamma = 0.9$, and equiprobable random policy

| | | | | |
|---|---|---|---|---|
| 3.3 | 8.8 | 4.4 | 5.3 | 1.5 |
| 1.5 | 3.0 | 2.3 | 1.9 | 0.5 |
| 0.1 | 0.7 | 0.7 | 0.4 | -0.4 |
| -1.0 | -0.4 | -0.4 | -0.6 | -1.2 |
| -1.9 | -1.3 | -1.2 | -1.4 | -2.0 |

# Understanding $V_\pi(s)$ with Gridworld



$$v_\pi(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right]$$

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] \\ &= \sum_a 0.25 \cdot \left[0 + 0.9 \cdot (2.3 + 0.4 - 0.4 + 0.7)\right] \\ &= 0.25 \cdot [0.9 \cdot 3.0] = 0.675 \approx 0.7 \end{aligned}$$

| | | | | |
|---|---|---|---|---|
| 3.3 | 8.8 | 4.4 | 5.3 | 1.5 |
| 1.5 | 3.0 | 2.3 | 1.9 | 0.5 |
| 0.1 | 0.7 | 0.7 | 0.4 | -0.4 |
| -1.0 | -0.4 | -0.4 | -0.6 | -1.2 |
| -1.9 | -1.3 | -1.2 | -1.4 | -2.0 |

# Ex-1

**Recollect the reward function used for Gridworld as below:**

- -1 if an action takes agent off the grid
- Exceptional reward from A and B for all actions taking agent to A' and B' resp.
- 0, everywhere else

**Let us add a constant c ( say 10) to the rewards of all the actions. Will it change anything?**

# Optimal Policies and Optimal Value Functions



- $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_\pi(s)$ for all $s \in S$
- There is always at least one policy that is better than or equal to all other policies → optimal policy (denoted as $\pi_*$)

**Optimal state-value function**

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

**Optimal action-value function**

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a)$$

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a]$$

- There could be more than one optimal policy !!!

# Optimal Policies and Optimal Value Functions



Bellman optimality equation - expresses that *the value of a state under an optimal policy must equal the expected return for the best action from that state*

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma v_*(s')\big].
\end{aligned}
$$

**Bellman optimality equation for $V_*$**

# Optimal Policies and Optimal Value Functions



Bellman optimality equation - expresses that *the value of a state under an optimal policy must equal the expected return for the best action from that state*

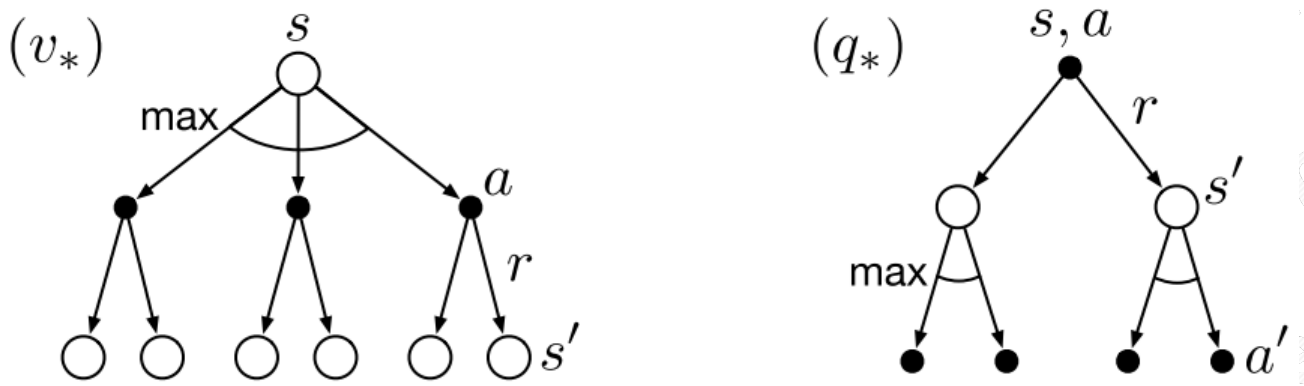
**Bellman optimality equation for $q_*$**

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\middle|\; S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right].
\end{aligned}
$$

# Optimal Policies and Optimal Value Functions

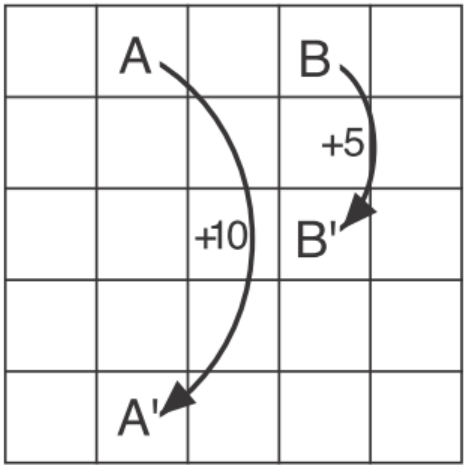


Bellman optimality equation - expresses that *the value of a state under an optimal policy must equal the expected return for the best action from that state*




Backup diagrams for v* and q*

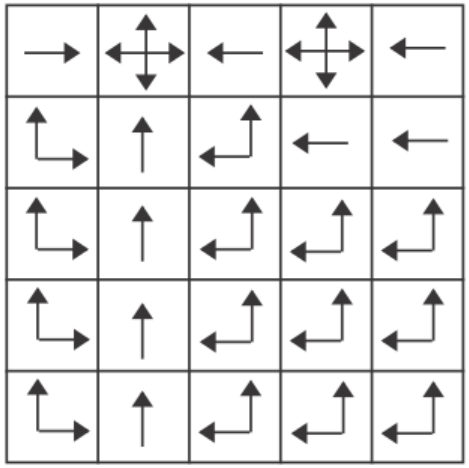
# Optimal solutions to the gridworld example




Gridworld

| | | | | |
|---|---|---|---|---|
| 22.0 | 24.4 | 22.0 | 19.4 | 17.5 |
| 19.8 | 22.0 | 19.8 | 17.8 | 16.0 |
| 17.8 | 19.8 | 17.8 | 16.0 | 14.4 |
| 16.0 | 17.8 | 16.0 | 14.4 | 13.0 |
| 14.4 | 16.0 | 14.4 | 13.0 | 11.7 |

$v_*$



$\pi_*$

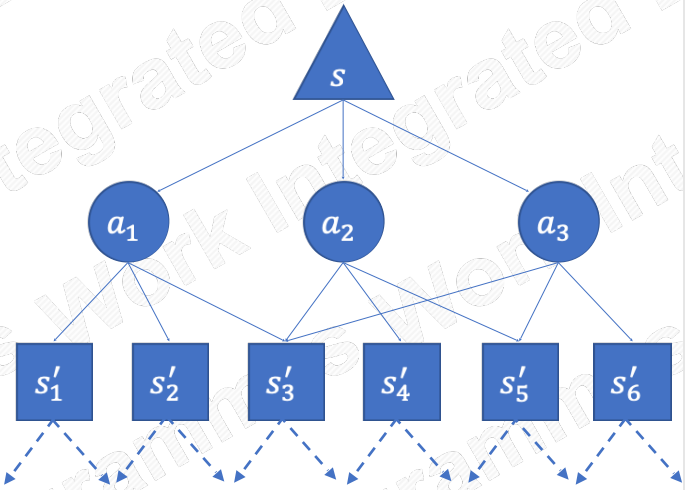
Backup diagrams for v* and q*

# MDP - Objective

- Compute a policy: what action to take at each state
  - $\pi: S \rightarrow A$

A set of states $s \in \mathcal{S}$
A set of actions $a \in \mathcal{A}$
State-transition probabilities $P(s'|s,a)$
A reward function $R(s,a,s')$

- Compute the **optimal** policy: maximum expected reward, $\pi^*$
- $\pi^*(s) = ?$
- $= \underset{a}{\text{argmax}}[\sum_{s'} P(s'|s,a)R(s,a,s')]$
  - Must also optimize over the future (next steps)
- $= \underset{a}{\text{argmax}}[\sum_{s'} P(s'|s,a)\,(R(s,a,s') + \underbrace{\mathbb{E}_{\pi^*}[G|s']}_{v^*(s')})]$



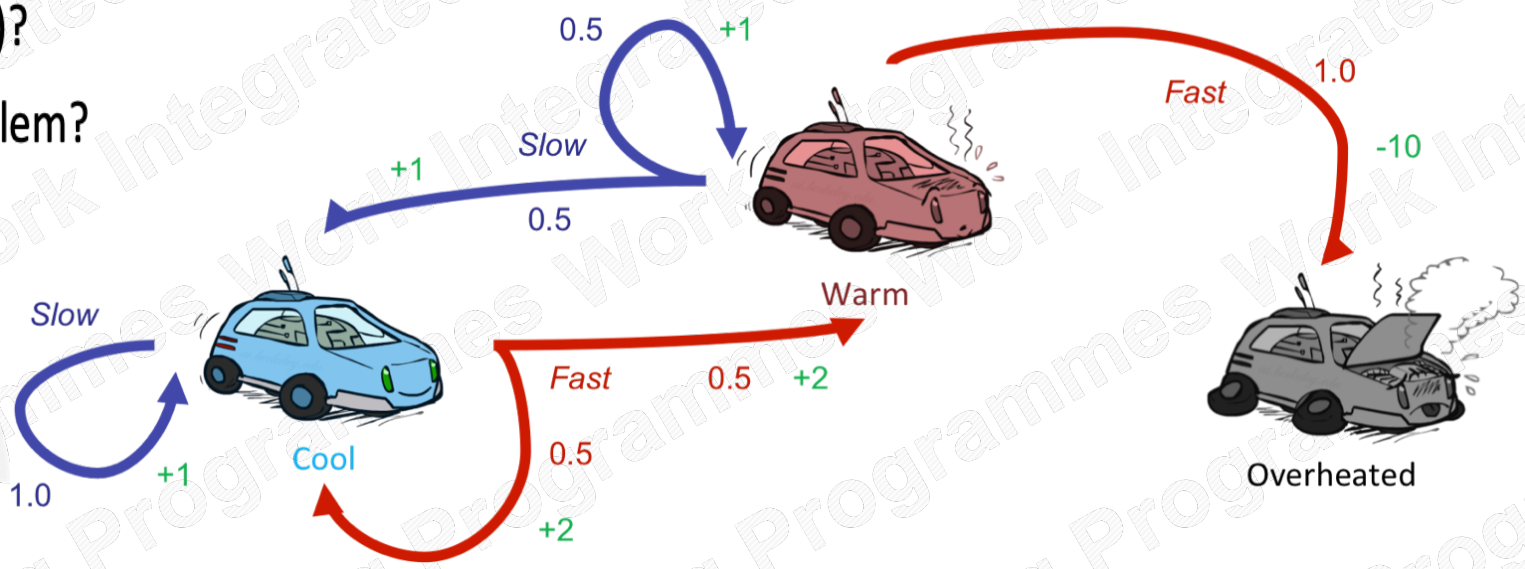
# Notation

- $\pi^*$ - a policy that yields the maximal expected sum of rewards
- $G$- observed sum of rewards, i.e., $\sum r_t$
- $v^*(s)$ - the expected sum of rewards from being at $s$ then following $\pi^*$
  - $= \mathbb{E}_{\pi^*}[G|s]$

# Race car example

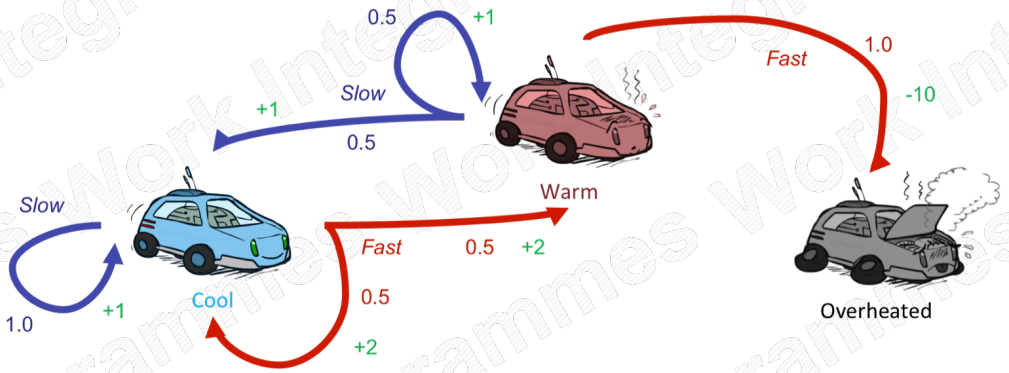
- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: *Slow*, *Fast*
- Going faster gets double reward
- What is $V^*(\text{Cool})$?
- What is $\pi^*(\text{Cool})$?
- What's the problem?

$$\max_a \left[ \sum_{s'} P(s'|s,a) \left( R(s,a,s') + v^*(s') \right) \right]$$



# Race car example

- Consider a discount factor, $\gamma = 0.9$
- What is $v^*(Cool)$
- $= \max_a \left[ r(s,a) + \sum_{s'} p(s'|s,a)\ \gamma\ v^*(s') \right]$
- $= \max[1 + 0.9 \cdot 1 v^*(Cool), 2 + 0.9 \cdot 0.5 v^*(Cool) + 0.9 \cdot 0.5 v^*(Warm)]$
  - Computing...
  - ...Stack overflow
- Work in iterations



# Value iteration



**Value iteration**

Initialize array $V$ arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

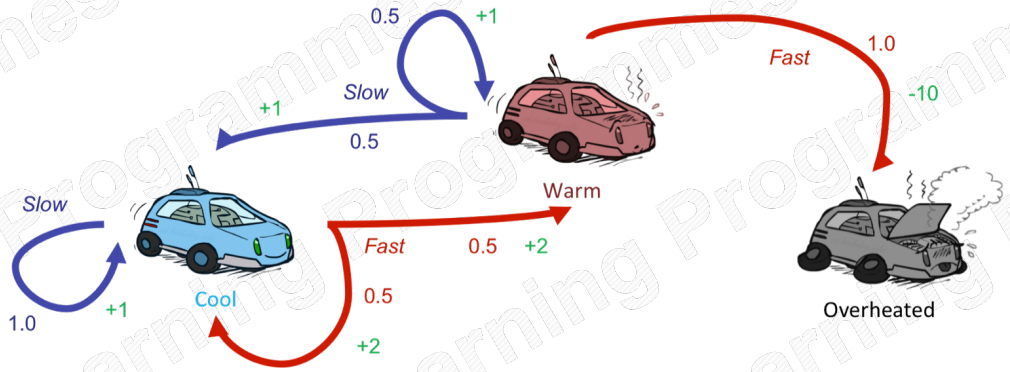
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

# Value Iteration

| | | | |
|---|---|---|---|
| $V_0$ | 0 | 0 | 0 |
| $V_1$ | 2 | 1 | 0 |
| $V_2$ | 3.35 | 2.35 | 0 |

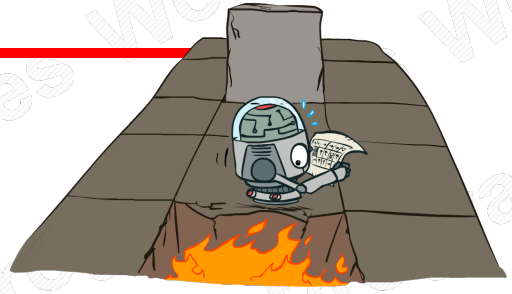
**Check this computation on paper.**



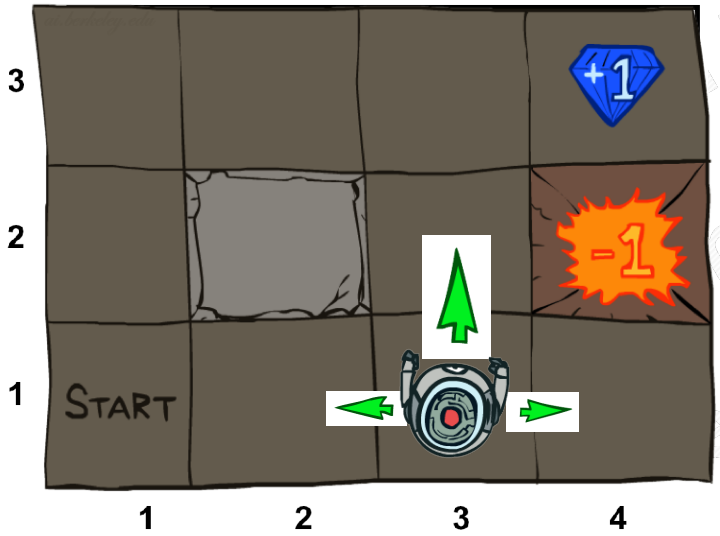

$\gamma = 0.9$

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_k(s')\right]$$

# Example: Grid World

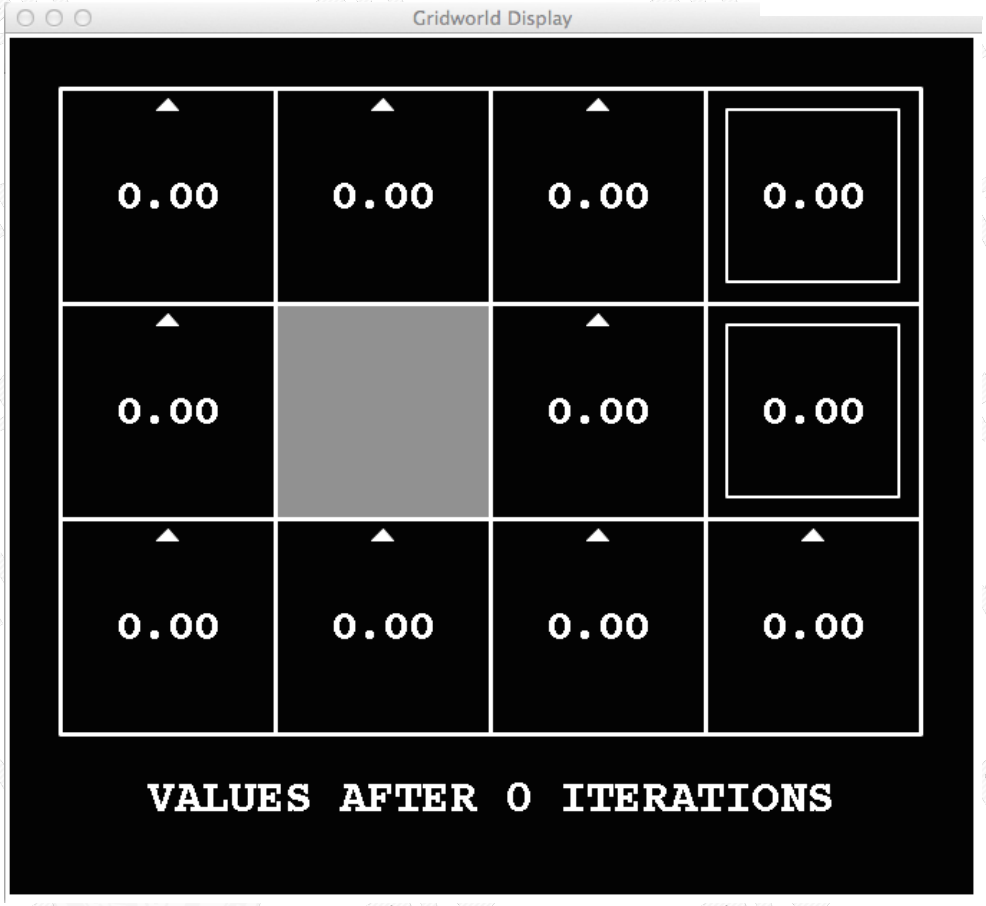


- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent's path
- Noisy movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - Small negative reward each step (battery drain)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of (discounted) rewards



# k=0

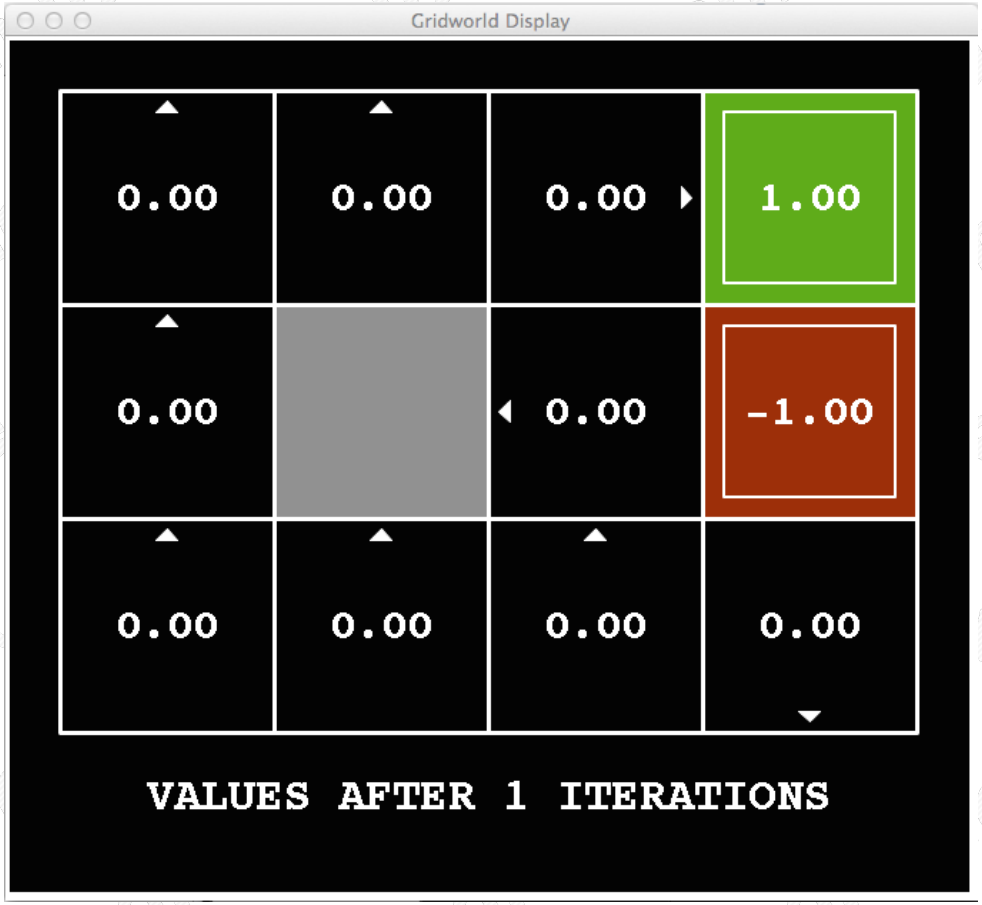
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_k(s')\right]$$




Noise = 0.2
Discount = 0.9
Living reward = 0

# k=1

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$




Noise = 0.2
Discount = 0.9
Living reward = 0

# k=2

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=3

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$

Noise = 0.2
Discount = 0.9
Living reward = 0

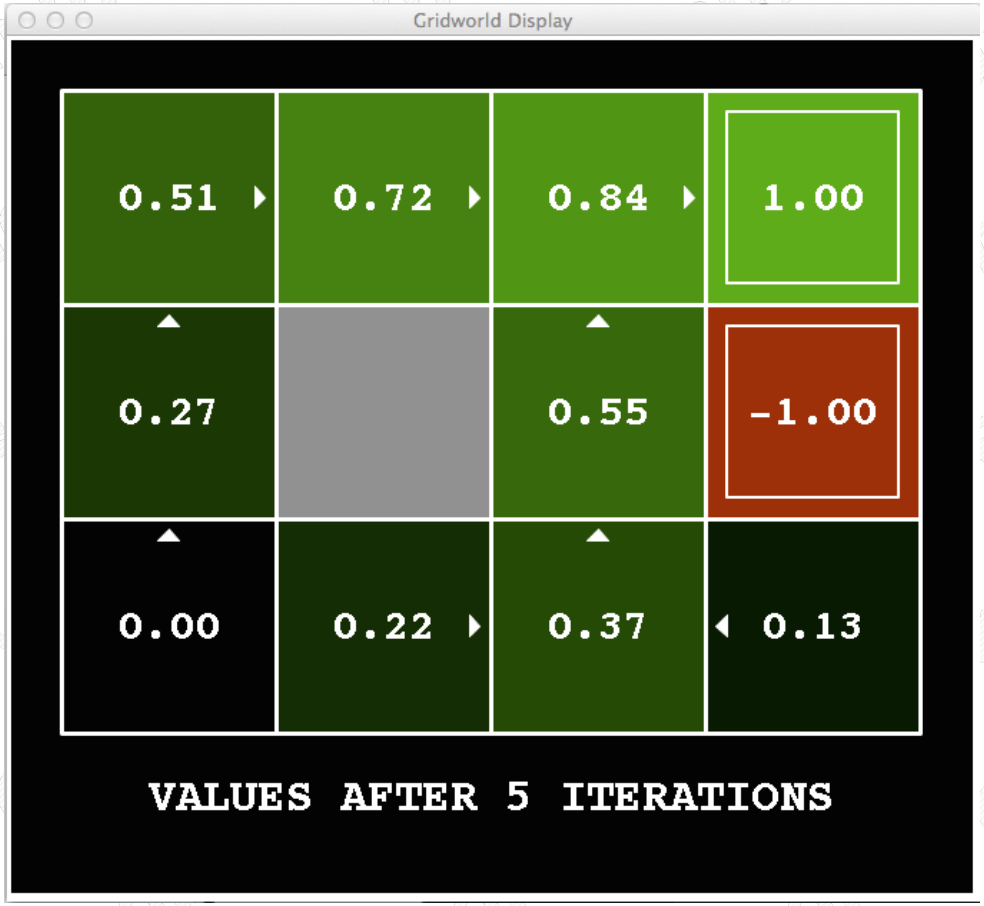
# k=4

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=5

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$



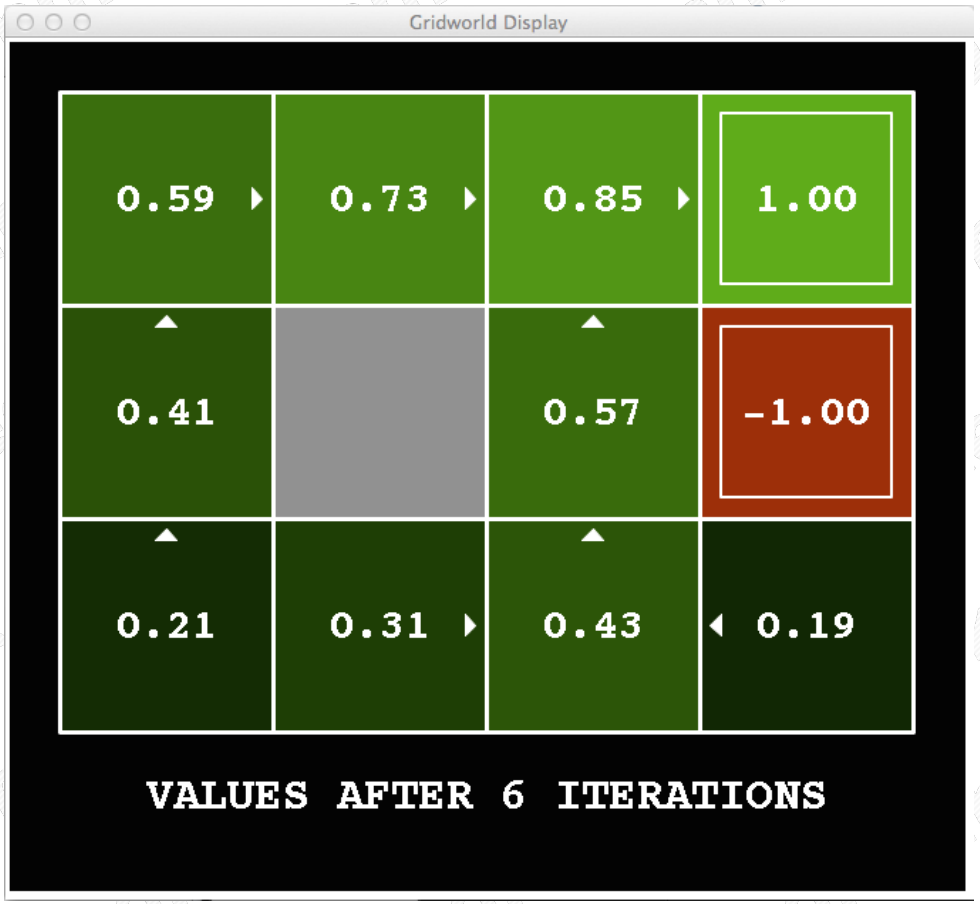

Noise = 0.2

Discount = 0.9

Living reward = 0

# k=6

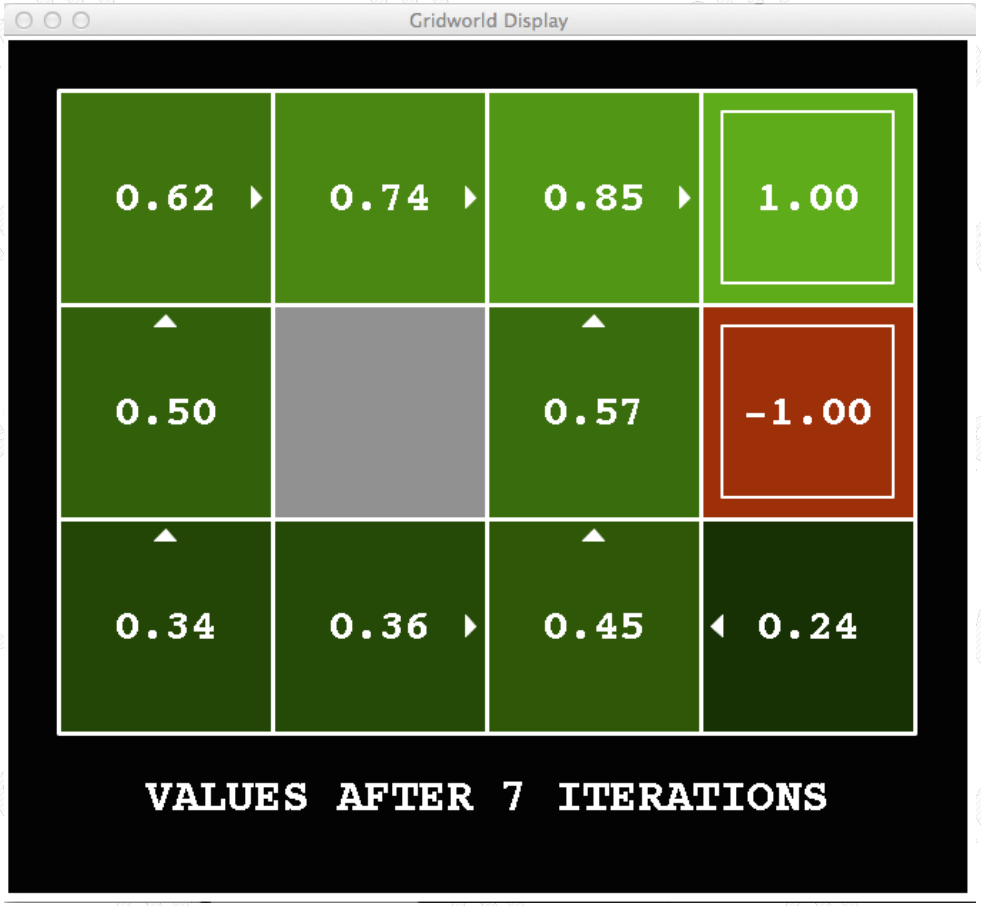
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$




Noise = 0.2
Discount = 0.9
Living reward = 0

# k=7

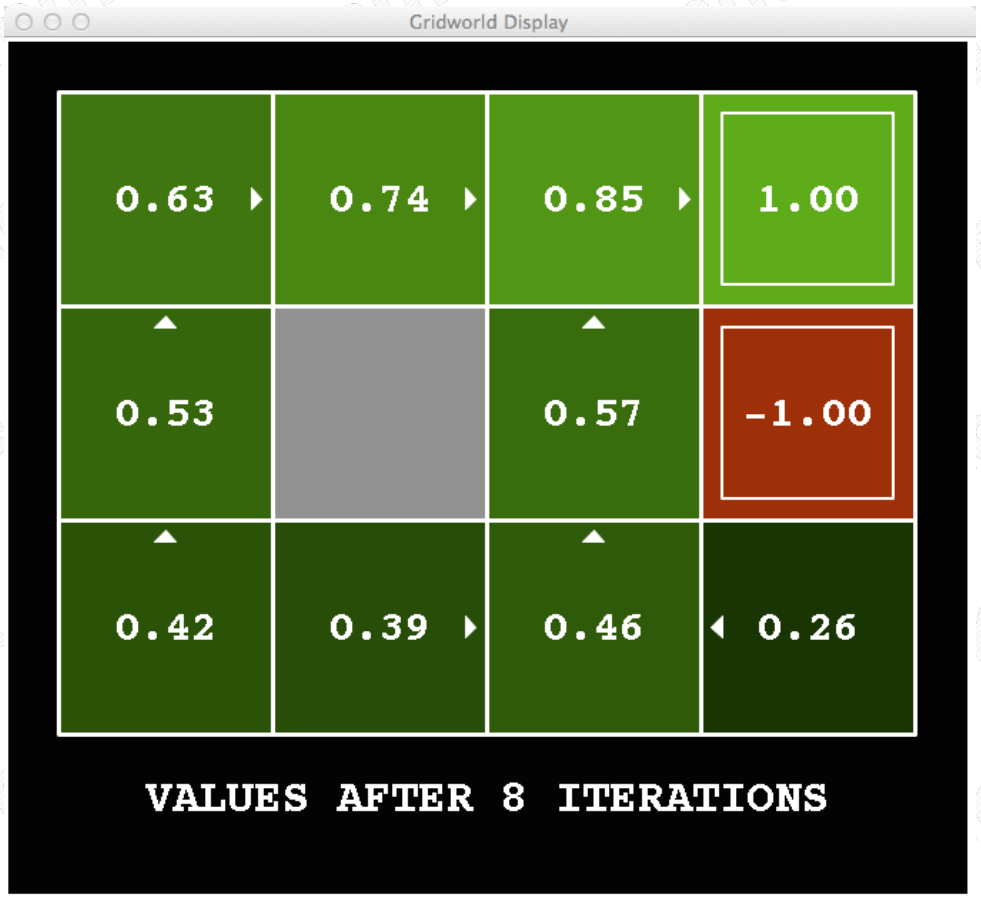
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$




Noise = 0.2
Discount = 0.9
Living reward = 0

# k=8

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$




Noise = 0.2
Discount = 0.9
Living reward = 0

# k=9

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

Noise = 0.2

Discount = 0.9

Living reward = 0

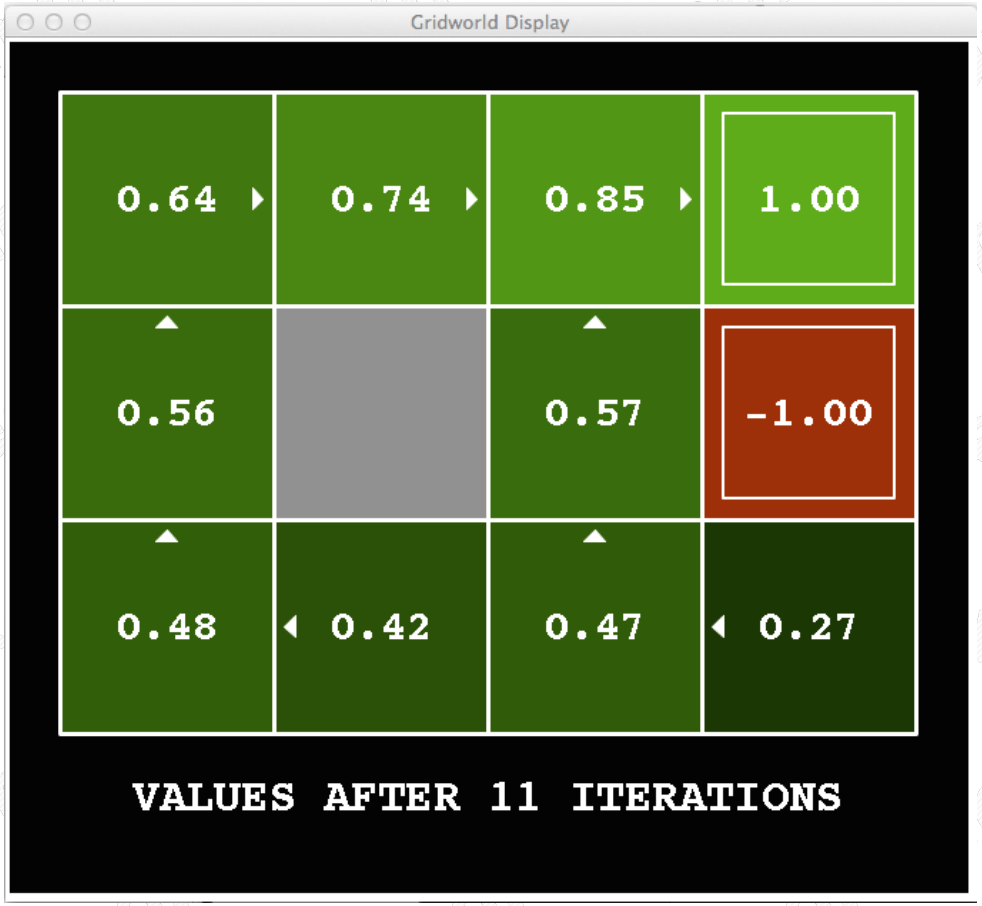
# k=10

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=11

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$




Noise = 0.2
Discount = 0.9
Living reward = 0

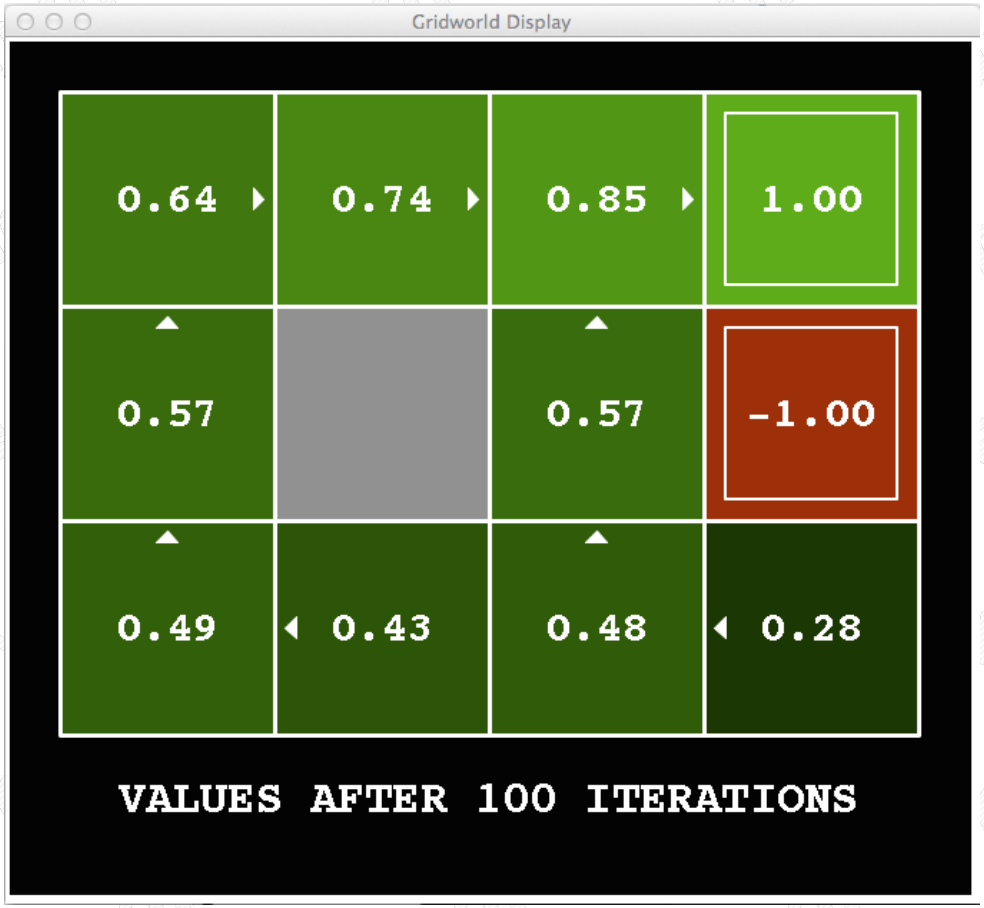
# k=12

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$

Noise = 0.2
Discount = 0.9
Living reward = 0

# k=100

$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$




Noise = 0.2
Discount = 0.9
Living reward = 0

# Problems with Value Iteration



- Value iteration repeats the Bellman updates:
- $V_{k+1}(s) \leftarrow \max_a \left[R(s,a) + \sum_{s'} P(s'|s,a)\ \gamma V_k(s')\right]$

$$= \max_a \left[\sum_{s'} P(s'|s,a)\ (R(s,a,s') + \gamma V_k(s'))\right]$$

- Issue 1: It's slow – $O(S^2A)$ per iteration
  - Do we really need to update every state at every iteration?
- Issue 2: A policy cannot be easily extracted
  - Policy extraction requires another $O(S^2A)$
- Issue 3: The policy often converges long before the values
  - Can we identify when the policy converged?
- Issue 4: requires knowing the model, $P(s'|s,a)$, and the reward function, $R(s,a)$
- Issue 5: requires discrete (finite) set of actions
- Issue 6: infeasible in large state spaces

# Solutions (briefly, more later…)



- Issue 1: It's slow – $O(S^2A)$ per iteration
  - Asynchronous value iteration
- Issue 2: A policy cannot be easily extracted
  - Learn *q* (action) values
- Issue 3: The policy often converges long before the values
  - Policy-based methods
- Issue 4: requires knowing the model and the reward function
  - Reinforcement learning
- Issue 5: requires discrete (finite) set of actions
  - Policy gradient methods
- Issue 6: infeasible for large (or continues) state spaces
  - Function approximators

# Issue 1: It's slow – $O(S^2A)$ per iteration

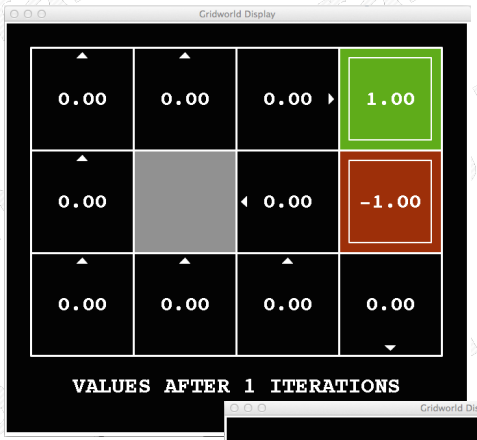


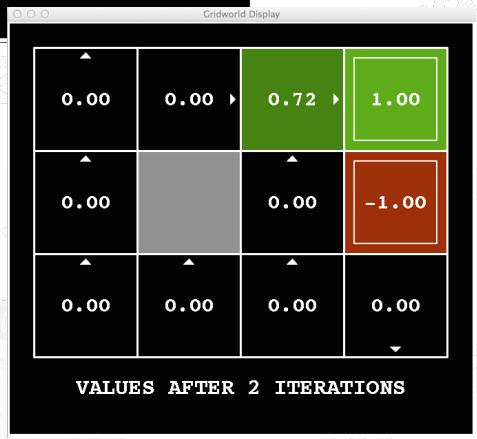
- Asynchronous value iteration
- In value iteration, we update every state in each iteration
- Actually, *any* sequences of Bellman updates will converge if every state is visited infinitely often regardless of the visitation order
- Idea: prioritize states whose value we expect to change significantly

# Asynchronous Value Iteration



- Which states should be prioritized for an update?




A single update per iteration

**Algorithm 3** Prioritized Value Iteration

1: **repeat**
2: $s \leftarrow \arg\max_{\xi \in S} H(\xi)$
3: $V(s) \leftarrow \max_{a \in A} \{R(s,a) + \gamma \sum_{s' \in S} Pr(s'|s,a)V(s')\}$
4: **for all** $s' \in SDS(s)$ **do**
5: // recompute $H(s')$
6: **end for**
7: **until** convergence

$$SDS(s) = \{s' : \exists a, p(s|s',a) > 0\}$$

For the home assignment set:

$$H(s') = \left| V(s') - \max_{a} \left\{ R(s',a) + \gamma \sum_{s''} \Pr(s''|s',a)V(s'') \right\} \right|$$

# Double the work?

For the home assignment set:

$$H(s') = \left| V(s') - \max_{a} \left\{ R(s', a) + \gamma \sum_{s''} \Pr(s''|s', a) V(s'') \right\} \right|$$

- Computing priority is similar to updating the state value (W.R.T computational effort)
- Why do double work?
  - If we computed the priority, we can go ahead and update the value for free
- Notice that we don't need to update the priorities for the entire state space
- For many of the states the priority doesn't change following an updated value for a single state $s$
- Only states $s'$ with $\sum_a p(s'|s,a) > 0$ require update

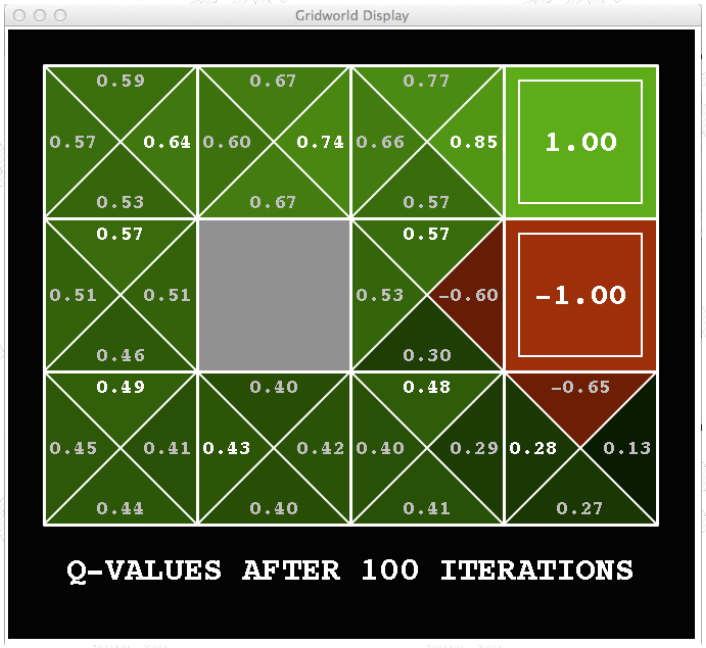
# Issue 2: A policy cannot be easily extracted



- Given state values, what is the appropriate policy?
  - $\pi(s) \leftarrow \underset{a}{\text{argmax}}\left[R(s,a) + \sum_{s'} P(s'|s,a)\ \gamma V_k(s')\right]$
  - Requires another full value sweep: $O(S^2A)$
- Learn $q$ (action) values instead
- $Q^*(s,a)$ - the expected sum of rewards from being at $s$, taking action $a$ and then following $\pi^*$

# Q-learning

- $Q^*(s,a)$ - the expected sum of rewards from being at $s$, taking action $a$ and then following $\pi^*$
- $\pi^*(s) \leftarrow \underset{a}{\text{argmax}}[Q^*(s,a)]$
- Can we learn Q values with dynamic programming?
  - Yes, similar to value iteration



# Q-learning as value iteration

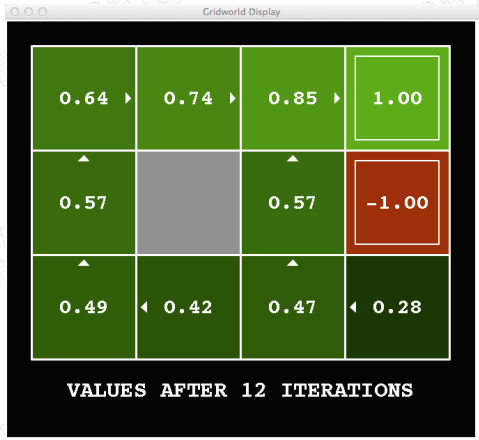


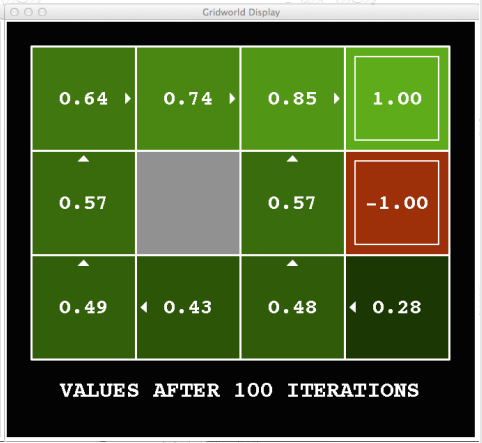
- 
- $V^*(s) \coloneqq \max_a [\sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma V^*(s'))]$
- $V^*(s) \coloneqq \max_a [Q^*(s,a)]$
- $Q^*(s,a) \coloneqq \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma V^*(s'))$
- $Q^*(s,a) = \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma \max_a [Q^*(s',a)])$
- Solve iteratively
  - $Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma \max_a [Q_k(s',a)])$
  - Can also use Asynchronous learning

# Issue 3: The policy often converges long before the values



- Value iteration converges to the true utility value: $V_{k\to\infty} \to V^*$
- $V^*$ implies the optimal policy: $\pi^*$
- Can we converge directly on $\pi^*$?
  - Improve the policy in iteration until reaching the optimal one

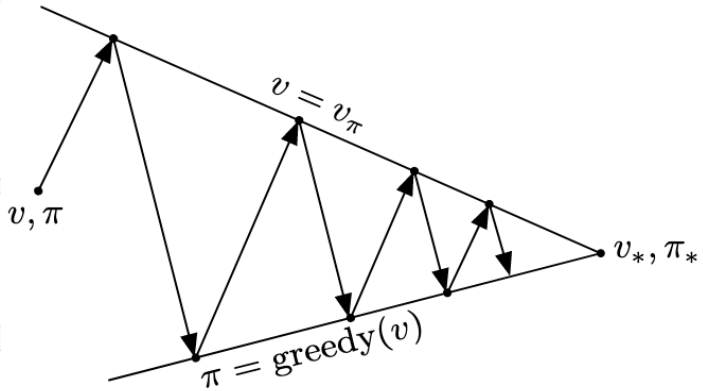


# Policy Iteration

1. Compute $V_\pi$: calculate state value for some fixed policy (not necessarily the optimal values, $V_\pi \neq V^*$)
2. Update $\pi$: update policy using one-step look-ahead with the resulting (non optimal) values
3. Repeat until policy converges

   (optimal values and policy)




- Guaranteed converges to $\pi^*$
  - $\forall s, V_{k>0}(s) \leq V_{k+1}(s)$ i.e., $\pi_i$ improves monotonically with i
  - A fixed point, $\forall s, V_k(s) = V_{k+1}(s)$, implies $\pi^*$

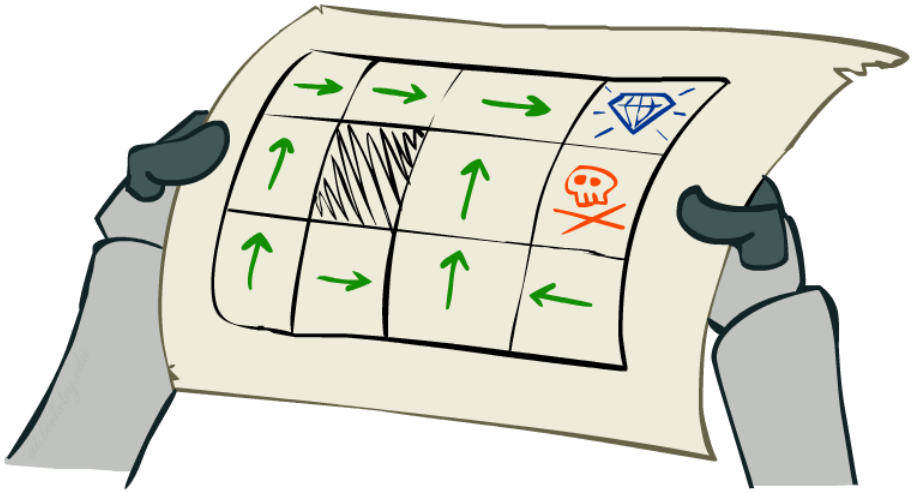
# Policy Evaluation

- Why is calculating $V_\pi$ easier that calculating $V^*$?
  - Turns non-linear Bellman equations into linear equations
- $v^*(s) = \max_a [\sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma v^*(s'))]$
- $v_\pi(s) = \sum_{s'} P(s'|s,\pi(s))\,(R(s,\pi(s),s') + \gamma v^*(s'))$
- Solve a set of linear equations in $O(S^2)$
  - Solve with Numpy (numpy.linalg.solve)
  - Required for your home assignment
  - See: https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html#numpy.linalg.solve

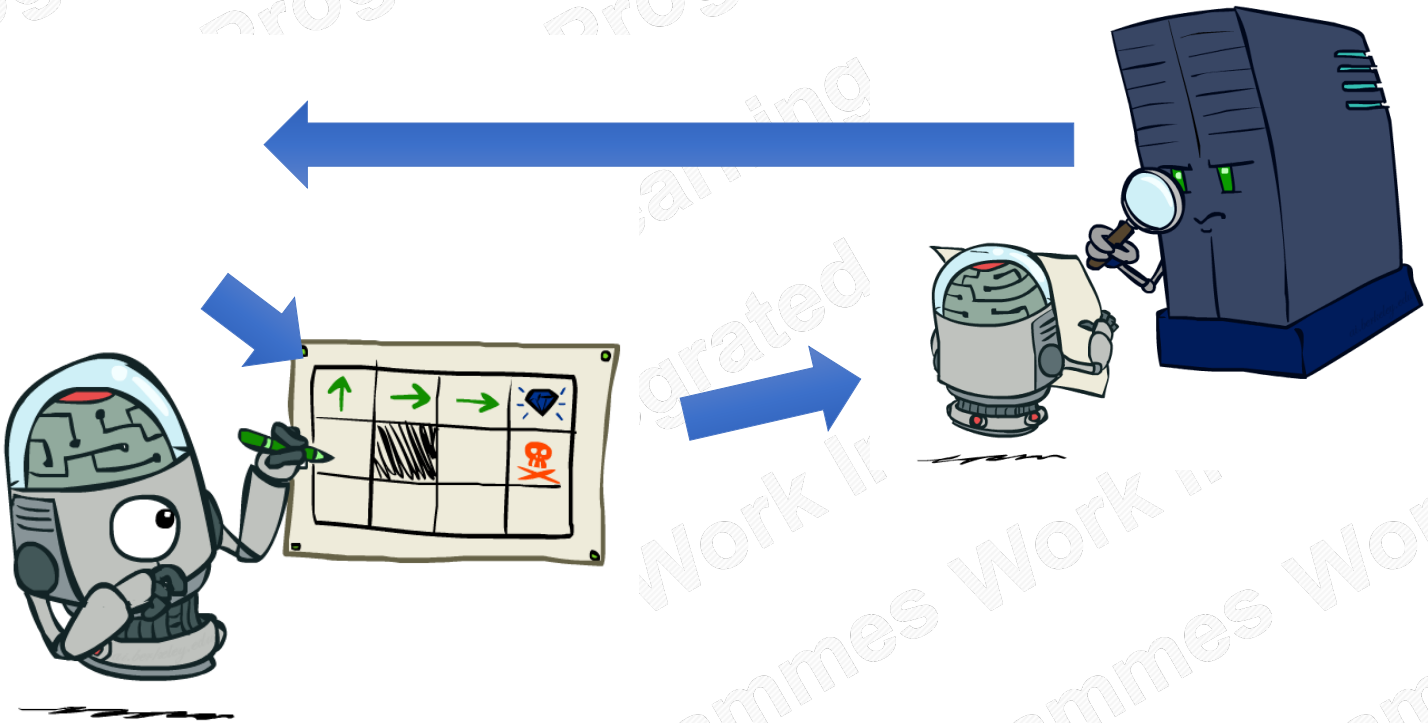
# Policy value as a Linear program



- $v_{11} = 0.8 \cdot (-0.1 + 0.95 \cdot v_{12}) + 0.1 \cdot (-0.1 + 0.95 \cdot v_{21}) + 0.1 \cdot (-0.1 + 0.95 \cdot v_{11})$
- $v_{12} = 0.8 \cdot (-0.1 + 0.95 \cdot v_{13}) + 0.2 \cdot (-0.1 + 0.95 \cdot v_{12})$
- ...
- $v_{42} = -1$
- $v_{43} = 1$



# Policy iteration



# Comparison

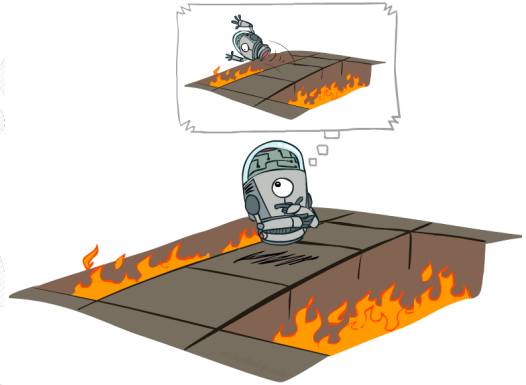


- Both value iteration and policy iteration compute the same thing (optimal state values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don't track the policy, but taking the max over actions implicitly define it
- In policy iteration:
  - We do several passes that update utilities with fixed policies (each pass is fast because we consider only one action, not all of them)
  - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  - The new policy will be better (or we're done)

# Issue 4: requires knowing the model and the reward function



- We will explore online learning (reinforcement learning) approaches
- How can we learn the model and reward function from interactions?
- Do we even need to learn them? Can we learn $V^*, Q^*$ without a model?
- Can we do without $V^*, Q^*$ ? Can we run policy iteration without a model?



Offline optimization

Online Learning

# Issue 5: requires discrete (finite) set of actions



- We will explore policy gradient approaches that are suitable for continuous actions, e.g., throttle and steering for a vehicle
- Can such approaches be relevant for discrete action spaces?
  - Yes! We can always define a continues action space as a distribution over the discreate actions (e.g., using the softmax function)
- Can we combine value-based approaches and policy gradient approaches and get the best of both?
  - Yes! Actor-critic methods

# Issue 6: infeasible in large (or continues) state spaces



- Most real-life problems contain very large state spaces (practically infinite)
- It is infeasible to learn and store a value for every state
- Moreover, doing so is not useful as the chance of encountering a state more than once is very small
- We will learn to generalize our learning to apply to unseen states
- We will use value function approximators that can generalize the acquired knowledge and provide a value to any state (even if it was not previously seen)

# Notation

- $\pi^*$ - a policy that that yields the maximal expected sum of rewards
- $V^*(s)$ - the expected sum of rewards from being at $s$ then following $\pi^*$
- $V_\pi(s)$ - the expected sum of rewards from being at $s$ then following $\pi$
- $Q^*(s,a)$ - the expected sum of rewards from being at $s$, taking action $a$ and then following $\pi^*$
- $Q_\pi(s,a)$ - the expected sum of rewards from being at $s$, taking action $a$ and then following $\pi$
- $G_t$ - observed sum of rewards following time $t$, i.e., $\sum_{k=t}^{T} r_k$

# Required Readings

1. Chapter-3,4 of Introduction to Reinforcement Learning,2nd Ed., Sutton & Barto

# Thank You…

BITS Pilani
Pilani Campus